Type Evolution and Instance Adaptation*

Stewart M. Clamen†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213-3890
U.S.A.

March 2, 1992

Abstract

Schema evolution support is an important facility for object-oriented database (OODB) systems. While existing OODB systems provide for limited forms of evolution, including modification to the database schema and reorganization of affected instances, we find their support insufficient. Specific deficiencies are 1) the lack of compatibility support for old applications, and 2) the lack of ability to install arbitrary changes upon the schema and database.

This paper examines the limitations of existing schemes, and offers a more general framework for specifying and reasoning about the evolution of class definitions and the adaptation of existing, persistent instances to those new definitions.

Keywords: Schema evolution, object-oriented databases, class versioning, instance adaptation, compatibility

*This research was sponsored in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; and by the Office of Naval Research under Contract N00014-88-K-0641. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

†Author can be reached via email at *clamen@cs.cmu.edu*, via phone at +1 (412) 268-2145, and via facsimile at +1 (412) 681-5739.

Introduction

Database systems exist to support the long-term persistence of data. It is natural to expect that, over time, needs will change and that those changes will necessitate a modification to the interface for the persistent data. In an object-oriented database (OODB) system, such a situation would motivate an evolution of the database schema. For this reason, support for **schema evolution** is a required facility in any serious OODB system.

An OODB database schema consists of class definitions and an inheritance hierarchy. A class is a tuple of methods and attributes. The database is populated by instances of those classes, with values for each of the attributes. The schema describes the interface between the set of application programs and the persistent repository of objects. When the schema changes so does the interface, possibly leaving incompatible elements on both sides of the barrier. We are interested in the problem of managing existing database objects that we call the **instance adaptation problem.** This paper examines the limitations of existing adaptation schemes and offers a more general framework for specifying and reasoning about the evolution of class definitions and the adaptation of existing, persistent instances to those new definitions.

Two general instance adaptation strategies have been identified and implemented by various OODB systems. The first strategy, **conversion,** restructures the affected instances to conform to the representation of their modified classes. Conversion is supported by the ORION[2, 13] and GemStone[5] systems.

The primary shortcoming of the conversion approach is its lack of support for **program compatibility.** By discarding the former schema, application programs that formerly interacted with the database through the changed parts of the interface are now obsolete. This is an especially significant problem when modification (or even recompilation) of the application program is impossible (*e.g.*, commercial software).

Rather than redefining the schema and converting the instances, the second strategy, **emulation,** is based on a **class versioning scheme.** Each class evolution defines a new version of the class, and old class definitions persist indefinitely. Instances and applications are associated with a particular version of a class, and the runtime system is responsible for simulating the semantics of the new interface on top of instances of the old, or vice versa. Since the former schema is not discarded but retained as an alternate interface, the emulation scheme provides program compatibility. Such a facility has been developed for the Encore system. [18]

Encore pays for this additional functionality with a loss in runtime efficiency. Under a conversion scheme, the cost of the evolution is a function of the number of affected instances. Once converted, an old instance can be referenced at the same cost as a

newly-created one. However, the cost of emulation is paid whenever there is a version conflict between the application and a referenced instance.

We feel however that program compatibility among schema versions is a very desirable feature. It can be of great utility in situations where the database is shared by a variety of applications, as in Computer-Aided Design or Office Automation Systems, when the database acts as a common repository for information, accessed by a variety of applications.

Our scheme supports program compatibility by maintaining multiple versions of the database scheme. Old programs can continue to interact with the database (on both new instances and old) using the former interface. Rather than emulating the evolved semantics all at runtime, efficiency is gained by representing each object as an instance of each version of its class. In this manner, our system affects a compromise between the functionality of emulation and the efficiency of conversion.

Another failing common to the conversion-based evolution facilities is the limitations placed on the variety of schema evolutions that can be performed. Most of the existing systems restrict admissible evolutions to a predefined list of schema[1] change operations (*e.g.*, adding/deleting an attribute or method from a class, altering a class's inheritance list). The length of this list might vary from system to system, but they are all similar in the way they support change: *The set of changes that can be performed are those which require either a fixed conversion of existing instances or no instance conversion at all.* Unfortunately, change is inherently unpredictable. A desired evolution is sometimes *revolutionary* and under such circumstances, these systems prevent the database programmer from performing the desired changes.

We are interested in supporting evolution in a liberal rather than a conservative fashion; rather than the system offering a list of possible evolutions to the programmer, the programmer should be able to specify arbitrary evolutions and rely on the system for assistance and verification. Change is a natural occurrence in any engineering task, and engineering-support systems should help rather than hinder when an evolution is required.

Encore's emulation facility restricts the breadth of class evolution that can be installed, but the restrictions are of a different form. Since instances, once created, cannot change their class-version, evolutions that require additional storage to be associated with each instance cannot be defined. (*cf.*, p.5 for details.)

[1]Throughout this paper, *schema* refers to the complete collection of class definitions and *class* refers to one a particular type.

newly-created one. However, the cost of emulation is paid whenever there is a version conflict between the application and a referenced instance.

We feel however that program compatibility among schema versions is a very desirable feature. It can be of great utility in situations where the database is shared by a variety of applications, as in Computer-Aided Design or Office Automation Systems, when the database acts as a common repository for information, accessed by a variety of applications.

Our scheme supports program compatibility by maintaining multiple versions of the database scheme. Old programs can continue to interact with the database (on both new instances and old) using the former interface. Rather than emulating the evolved semantics all at runtime, efficiency is gained by representing each object as an instance of each version of its class. In this manner, our system affects a compromise between the functionality of emulation and the efficiency of conversion.

Another failing common to the conversion-based evolution facilities is the limitations placed on the variety of schema evolutions that can be performed. Most of the existing systems restrict admissible evolutions to a predefined list of schema[1] change operations (*e.g.*, adding/deleting an attribute or method from a class, altering a class's inheritance list). The length of this list might vary from system to system, but they are all similar in the way they support change: *The set of changes that can be performed are those which require either a fixed conversion of existing instances or no instance conversion at all.* Unfortunately, change is inherently unpredictable. A desired evolution is sometimes *revolutionary* and under such circumstances, these systems prevent the database programmer from performing the desired changes.

We are interested in supporting evolution in a liberal rather than a conservative fashion; rather than the system offering a list of possible evolutions to the programmer, the programmer should be able to specify arbitrary evolutions and rely on the system for assistance and verification. Change is a natural occurrence in any engineering task, and engineering-support systems should help rather than hinder when an evolution is required.

Encore's emulation facility restricts the breadth of class evolution that can be installed, but the restrictions are of a different form. Since instances, once created, cannot change their class-version, evolutions that require additional storage to be associated with each instance cannot be defined. (*cf.*, p.5 for details.)

[1] Throughout this paper, *schema* refers to the complete collection of class definitions and *class* refers to one a particular type.

In the remainder of this paper, we present a model for specifying schema evolutions and instance adaptation strategies. Our system supports program compatibility, accepts a larger variety of evolutions than existing systems, and supports a variety of options to make it more efficient than the pure emulation facility of Encore.

Related work

Before describing our system in detail, we present a more detailed description of important existing systems.

ORION

The most ambitious and effective example of a schema evolution support facility is that provided by ORION [2, 13, 14]. ORION provides a taxonomy of schema evolution operations. It also defines a database model in the form of invariants that must be preserved across any valid evolution operation and a set of rules that instruct the system how best to maintain those invariants. Under this model, a schema designer specifies an evolution in terms of the taxonomy, the system verifies the evolution by determining if it is consistent with the invariants and then adjusts the schema and database according to the appropriate rules.

ORION can only perform those evolutions for which it has a rule defined. The set of rules is fixed. For example, changes to the domain of an attribute of a class are restricted to generalizations of that domain. This restriction exists because there is no facility in ORION's evolution language for defining a procedure that to be used by the system to convert old instance values. Generalizations of the attribute domain are allowed since this evolution does not require existing instances to be modified.

In ORION, evolutions are performed on a unique schema. Instances are converted under a **lazy conversion** scheme; that is, they are not converted when the evolution is declared, but instead converted when they are referenced. Under this scheme, there is no compatibility support for old programs and, depending on the evolution, information contained in the instances might be lost at conversion time. (*e.g.*, Deletion of an attribute.)

GemStone

Like ORION, GemStone supports a set of evolution operations. It is distinguished by its employment of an **eager conversion** scheme, converting affected instances when the evolution is specified. This scheme has the advantage that no runtime support is required or expense incurred; once installed, the restructuring of the database is complete. This is in contrast to lazy conversion, which requires the runtime system to check for the existence of still-unconverted instances indefinitely. On the other hand, eager conversion makes evolutions very time-consuming to install.

Encore

Encore implements emulation via user-defined exception handling routines. Whenever there is a version conflict between the program and the referenced instance, the routine associated with that method or instance (and those pair of versions) is called. The routine is expected to make the method's invocation conform to the expectations of the instance or make the return value from the method invocation consistent with the expectations of the calling program, whichever is appropriate. It is known, however, that certain evolutions cannot be modeled adequately under this scheme. The problem stems from the fact that each object can only instantiate a single version. If an evolution includes the addition (subtraction) of information (*e.g.,* the addition (deletion) of an attribute), there is no place for older (newer) instances to store an associated value. The best a programmer could do in such a system is associate a default attribute value for all instances of older (newer) type-versions by installing an exception handling routine to return the value when an application attempts to reference that attribute from an old (new) instance. [18]

The Common Lisp Object System

CLOS[19, 12], while not an OODB system, provides extended support for class evolution nonetheless. As Common Lisp system development is performed in an interactive context, class redefinition is a frequent occurrence. Rather than discard all existing instances, CLOS converts them according to a policy under the control of the user. The default policy is to reinitialize attribute values that no longer correspond to the attribute domain, and to delete attribute slots that are no longer represented in the class definition. Users can override this policy by defining their own method that is called automatically by the system. This method is passed as arguments the old and new slot values, so relationships between deleted and added attributes can be enforced.[19, p.859]

Adaptation and Extension of (Relational) Views

Bertino[4] presents a schema evolution language which is an OODB adaptation of the view mechanism found in many relational database systems. Her primary innovations are the support of inheritance and *object IDs* (OIDs) for view instances, two important characteristics of OODB models that are not present in the relational model. View instances with OIDs are physically realized in the database, enabling the view mechanism to support evolutions that specify the addition of an attribute, as envisioned by Zdonik[?1]. However, Bertino's scheme focuses on how evolutions affect the schema. It

is not concerned explicitly with the effects upon the instances nor with compatibility issues.

CLOSQL

Monk's CLOSQL[16] implements an class versioning scheme, but employs a conversion adaptation strategy. Instances are converted when there is a version conflict, but unlike ORION and GemStone, CLOSQL can convert instances to older versions of the class if necessary.

Lerner and OTGen

Lerner's OTGen design[15] addresses the problem of complex evolutions requiring major structural conversions of the database (*e.g.*, information moving between classes, sharing of data using pointers) using a special-purpose language to specify instance conversion procedures. As it was developed in an integrated database context, where the entire application set is recompiled when schema changes, versioning and compatibility were not considered.

Conversion and Compatibility

Schema Modification vs. Class Versioning

The schema evolution support provided by such systems as ORION and GemStone is restricted to what Kim calls **schema modification**, that is, *the direct modification of a single logical schema.* [14] When only one database schema exists, it is appropriate for the system to convert all existing instances. From a database consistency perspective, it must *appear*[2] that all instances have been converted when the evolution operation is applied. In fact, we would claim that it is the *only* sensible approach.

As has already been stated, however, conversion might render the instances inaccessible to applications that had previously referenced them. The adaptation strategy converts the instances but does not alter procedural references. Thus, application programs written and compiled under the old schema may now be obsolete, unable to access either the old, now converted, instances, or the ones created under the new schema.

A reasonable direction of research here would be to provide some automated mechanisms to assist with program conversion; it is an active line of research. [11, 1] In the OODB context, some work has been conducted at providing support to alert the programmer about the procedural dependencies of their evolution operation. [8] But this is not the only possible solution. Rather than adjust programs to conform to the data, it would seem easier to adjust the data to conform to the existing programs. Also, it is not always possible to alter, or even recompile, programs (*e.g.*, commercial software). This lack of compatibility support is our primary motivation for adopting a class versioning design for evolution management and support.

Under a class versioning scheme, multiple interfaces to a class, one per version, are defined. When compiled, application programs are associated with a single version of each of the classes it refers to; a *schema configuration*, if you will. With the database populated with instances of multiple versions of a class, the runtime system must resolve discrepancies between the version expected by the application and that of the referenced instance.

Objects Instantiating Multiple Versions

Under the original Encore scheme[18], instances never change their type-version. Aware of the restrictions this causes (*cf.*, previous section), Zdonik proposed a scheme whereby an existing instance can be "wrapped" with extra storage and a new interface, enabling it to be a full-fledged instance of a new type-version. [21] While still able to be accessed through its original interface/version, the wrapped object can also be manipulated

[2] Whether the instances are converted eagerly or lazily becomes an implementation issue.

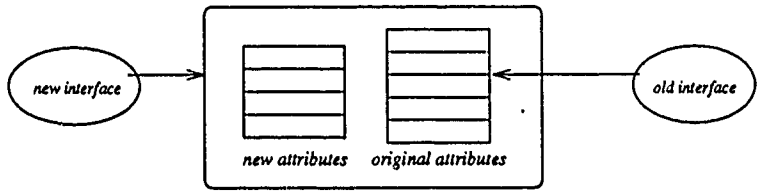


Figure 1: Zdonik's Wrapping Scheme: as in the Encore design, multiple interfaces to the class are preserved. Here, extra space is allocated for the attributes added as a result of the evolution, and applications can access the instance through either the old or new interfaces.

through the new interface. Thus, if the class evolution specifies the addition of an attribute, the wrapping mechanism could allocate storage for the new slot in existing instances, without denying backward compatibility. (*cf.*, Figure 1)

Our proposed scheme is a generalization of this approach. Much like each class has multiple versions, each instance is composed of multiple **facets**. Theoretically, these *multifaceted* instances encapsulate the state of the instance for all the defined interfaces (versions). The representation of these instance resembles a disjoint union of the representation of each of the versions, and it is useful to consider the representation as exactly that. As will be explained later, however, the process of actually allocating and initializing the facets can be deferred until needed.

As an example, consider a class Undergraduate, originally including attributes Name, Program, and Class, and a new version of the class with the attributes Name, Id Number, Advisor, and Class Year. (Class is one of {Freshman, Sophomore, Junior, Senior}, while Class Year is the year the student is expected to graduate.) Program is the degree program in which the student is enrolled, and Advisor is his academic advisor. While instances of Undergraduate in the database will contain all seven distinct attribute slots, any particular application will be restricted to one version and thus only have explicit access to one facet.

In reasoning about the relationship between any two versions[3] of a class, it is useful to divide the attributes into these four groups:

Shared: when an attribute is common[4] to both versions,

[3] For explanatory purposes, imagine that we are describing a class consisting of only two versions, and where the database is populated by instances of both.

[4] Common in the semantic sense, not just having the same name or type.

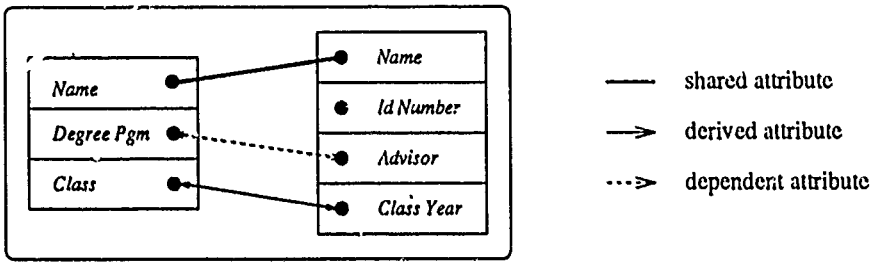


Figure 2: Disjoint union representation of the versioned class Undergraduate

Independent: when an attribute's value cannot be affected by any modifications to the attribute values in the other facet.

Derived: when an attribute's value can be derived directly from the values of the attributes in the other facet,

Dependent: when an attribute's value is affected by changes in the values of attributes in the other facet, but cannot be computed solely from those values.

In our example, the Name attribute is shared by the versions, while Id Number is independent. Class and Class Year are both derived attributes, since, given the current date, it is possible to derive one from the other. Advisor is a dependent attribute, since a change in Program might necessitate a change in advisor. Likewise, Program is a dependent attribute, since a change in advisor might imply that the student has switched degree programs. (*cf.*, Figure 2)

Zdonik *et al.* [18, 21] almost always cite evolutions involving independent or derived attributes in their examples. The original Encore emulation scheme is adequate for supporting evolutions that introduce shared and derived attributes. Zdonik's wrapping proposal addresses the problems associated with independent attributes. Our proposed scheme, however, will provide a mechanism for managing class evolutions that include dependent attributes. (See Table (p.11) for a comparison of the evolution capabilities of various systems.)

Classes, and thus class-versions, are made up of methods as well as attributes. Most object-oriented data models allow for the specification of private attributes that can only be manipulated by the methods of the class. With respect to class evolution, the addition, deletion, or alteration of a method that does not change the semantics of the component instances can only affect the database schema, and thus no adaptation of existing instances is required. For such changes, existing schema evolution technologies perform adequately.

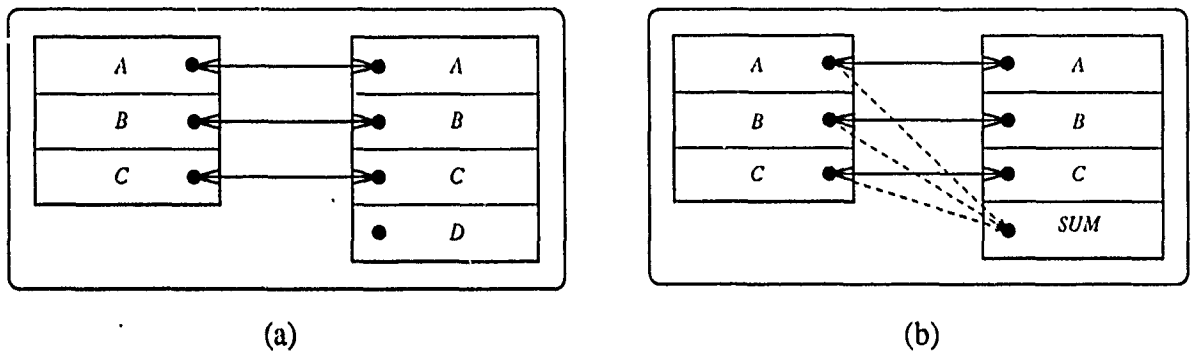


Figure 3: Some simple evolutions: (a) illustrates the relationships following the addition of an independent attribute, (b) shows the addition of a derived attribute.

Interacting with Multifaceted Instances (Example)

Consider an evolution specification language which can categorize attributes. This is accomplished by associating extra information with each attribute.

Shared attributes have the name of the corresponding attribute from another version,

Derived attributes have a function for determining its value in terms of the values of the attributes in the other version, and

Dependent attributes have a function for determining its value in terms of the values of the attributes in both versions (*i.e.*, the entire object), and

Independent attributes have nothing extra at all.

Representing the class instances as a disjoint union of the version facets, as described earlier, consistency between the facets can be maintained according to the following procedure:

> *Whenever an attribute value of a facet is modified, those attributes in the other facet that depend on it must be updated. For shared attributes, the new value is copied; for dependent and derived attributes, the dependency functions are applied and the result written into the (attribute) slot in the other facet.*

Description of Evolution	ORION	Encore	Bertino[4]	Us
Add/delete/rename a method	√	√	√	√
Add an attribute	√	×	√	√
Delete an attribute	√	√	√	√
Generalize the domain of an attribute	√	√	√[a]	√
Change (arbitrarily) the domain of an attribute	×	√	√[a]	√
Telescoping[b]	√	√	√	×[c]
Change supertypes	√	?	√	×

Table 1: Some evolutions and which systems support them.

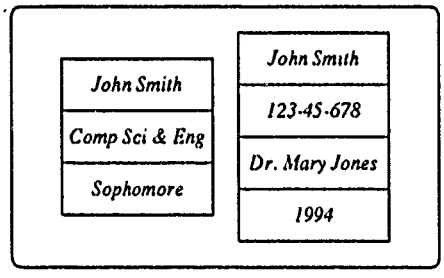
[a]This evolution cannot be define directly, but can be implementing by replacing the attribute in the new version with generic methods for reading and writing.[3]

[b]Definition appears in conclusions

[c]*cf.*, (p.16) for clarification.

The remainder of this section consists of an example.

Consider a multifaceted instance of the Undergraduate versioned class, represented graphically as follows:



Imagine that John Smith returns to university after his first summer vacation and wishes to change to the undergraduate Math program. Also, he had taken some summer classes that have given him enough credits to graduate a year early. The change to his data record are recorded through an application program employing the first version of the Undergraduate class. The system must now propagate those modifications to the second facet.

The attributes Class and Class Year can be derived in terms of each of other. A reasonable derivation function for Class Year is:

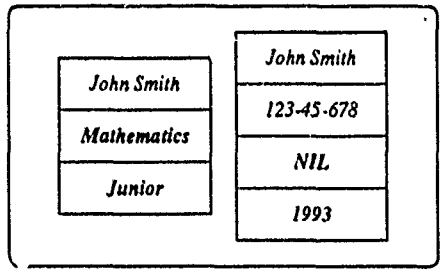
$$\text{ClassYear} = \begin{cases} \text{cy} + 3 & \text{if Year} = \texttt{Freshman} \\ \text{cy} + 2 & \text{if Year} = \texttt{Sophomore} \\ \text{cy} + 1 & \text{if Year} = \texttt{Junior} \\ \text{cy} & \text{if Year} = \texttt{Senior} \end{cases}$$

Where cy is the current year. The Advisor attribute is dependent upon the value of the Program attribute, but not completely derivable. A reasonable dependency function is:

$$\text{Advisor} = \begin{cases} \text{Advisor} & \text{if Advisor} \in \text{Program faculty} \\ \text{nil} & \text{otherwise} \end{cases}$$

Since there is not enough information to derive it, the student's advisor will have to be filled in later.

Applying these functions in concert with the desired changes to John Smith's record, the multifaceted instance becomes:



Representing Multifaceted Instances

In the previous section we described the semantics of our schema versioning scheme. In this section we address the issue of how to realize these multifaceted instances physically in the database.

Our basis for consideration is a system which implements the design as described: class evolutions are defined by creating a new version of the class; new facets (corresponding to the new version) are associated with every instance of the class and initialized according to a user-defined procedure. Each application program interacts with the instances through a single (interface) version and modifications to attribute

slots on the "visible" facet are immediately propagated to the other facets, using a mechanism similar to the trigger facility found in many relational and AI database systems [20, 10].

The most obvious target for improvement in this scheme is how new facets are added. The allocation and initialization of new facets for existing instances at evolution time is subject to some of the same criticisms as eager conversion (*cf.*, p.4). Thus, it is advantageous to defer the addition of the new facet as long as possible, *i.e.*, until an application program attempts to reference the new facet.

The strategy of deferring the actual maintenance of a dependency constraint until its effect is actually required can be applied as well to the propagation of information among the facets of an instance. Rather than update the attribute values of the other facet(s) each time a facet attribute is modified, one need only bring a facet up-to-date when there is an attempt to access it. This scheme can be supported by associating a flag with each facet indicating whether the facet is up-to-date with respect to the most recently modified facet. Read operations on facets with an unset flag are preceded by a resynchronization operation, which performs any necessary updates and sets the flag.

This scheme reduces overall runtime expense, since the resynchronization step is not performed in concert with every update operation, as was previously the case. However, it does increase the potential cost of inexpensive read operations.

To this point, we have been very liberal with our allocation of space for instance representation. Although the lazy allocation of facets conserves some space in the short run, the disjoint union model implies that every instance of a versioned class will have a complete collection of facets. There are a few optimizations that could be performed to reduce space requirements.

The first space-saving improvement entails having each set of shared attributes occupy a single slot in the multifaceted version. A performance improvement might also be realized here, since slot sharing reduces the expense and/or frequency of update propagations. (*cf.*, Figure 4 (p.14).)

Under certain circumstances, the slot associated with a derived attribute can be recovered as well. If an inverse procedure to the derivation function is known to the system, then the attribute can be implemented using the appropriate reader and writer methods to simulate it. For many evolutions, the inverse procedure appears as the derivation function for the related attribute in the other facet. The Year and Class Year attributes in our example (p.8) are related in that way. (*cf.*, Figure 5 (p.14).)

From a runtime performance perspective, this space optimization reduces the expense of write operations while making read operations more costly. The slot allocated for a derived attribute acts as a cache for its derivation function and, depending on

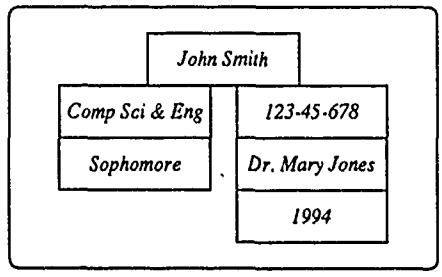


Figure 4: Multifaceted instance representation using common slot for shared attributes

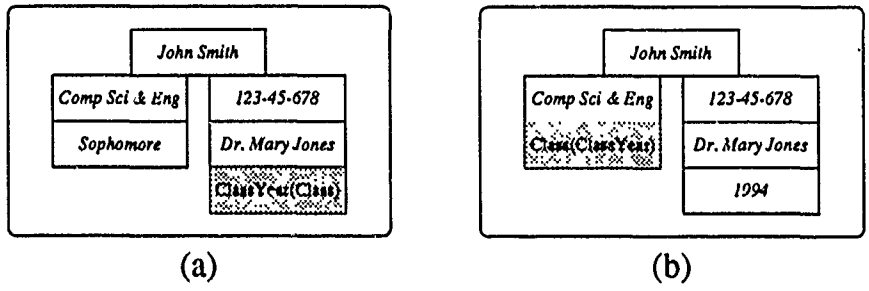


Figure 5: Multifaceted instance representation minimizing derived attribute allocation. For the Undergraduate class, two minimizations exist.

the frequency of modifications to its dependent attributes in the other facet(s), its maintenance might be more time-efficient.

Forcing an Adaptation Strategy

While an important feature in general, program compatibility is not always required (*e.g.*, a database with a single application program and a single user). In such situations one should be able to improve performance by instructing the system to convert fully the existing instances and discard (or perhaps archive) the old information. Furthermore, conversion and compatibility are not mutually exclusive. As long as an inverse conversion procedure is known, one could convert and emulate the former version. This might be useful when you want to preserve compatibility, but expect that it will be needed infrequently enough that you are willing to pay the cost of emulation in those instances. If it is the case that an applications then to access a distinct subset of the

instance collection, one could use a strategy that converts (on access) instances to the version of the application. (This is the approach taken by Monk's CLOSQL system[16].)

Sometimes, modification of the database or its schema is impossible. Databases might be read-only for permission (*e.g.*, remote database exported as a public service) or licensing reasons (*e.g.*, reference materials on CD-ROM). In such situations, something resembling Zdonik's wrapping scheme (*cf.*, p.7) must be used, with the wrapper actually residing in a separate database.

Inheritance

When a class is evolved, it may not only its direct instances, but the instances of its subclasses as well. If we consider class instances as represented by slices, each slice instantiating the superclass, then it is easy to see what would happen. The instance adaptation scheme that applies to direct instances applies to the appropriate slices in each of the subclass instances.

However, that does not explain how our system handles evolutions that affect the inheritance graph of the schema. Changes to the inheritance list of a class (*i.e.*, addition or deletion of a superclass, or the reordering of the superclass list) can be viewed as a compound evolution, incorporating many additions and deletions of attributes. The mechanisms described for the addition or deletion of single attributes work similarly here.

Support for evolutions that merge or split existing classes into new classes is discussed in the Future Work section.

Conclusions and Future Work

In this paper, we have described a specification model for schema evolution that has the following features:

- Schema versioning instead of modification to a single schema, so that program compatibility can be supported, if desired.
- Compatibility support is provided at less runtime cost than the Encore facility. For each version of a class, each instance has a corresponding facet. For attributes which can be derived solely from attributes from other facets, this facility is like a cache, sacrificing space for time. For attributes which are not reflected in the representation of the versions, the facet provides space for the value to stored, thereby preserving information that would be lost under a conversion scheme.
- A broader variety of evolutions are supported than in existing systems (ORION, GemStone, Encore). However, not all evolutions, (*e.g.*, telescoping (see below) and the more complex reorganization evolutions of Lerner[15]), are currently possible. (See below)
- Fine tuning of the adaptation scheme is possible, by allowing the programmer to decide how much duplication of information is present in each instance. If one version is used very infrequently, the programmer can save space and time, by emulating its interface instead.

The remainder of this section discusses topics and goals for ongoing and future research.

Complex Evolutions involving Inheritance

There are some complex evolutions that we have not addressed: *telescoping*, and class splitting.

A telescoping evolution is one that collects attributes from the component classes and installs them as attributes in the evolved class. [17] The problem our scheme has with these evolutions is that the derivation function in such a case refers to other instances whose value can change without the first instance being notified. In order to support this type of evolution, the programmer must be able to force the attribute to be derived each time it is referenced, and a generic write method would need to be supplied.

The problem associated with the act of splitting a class up is that it might involve a splitting of the instance collection as well. We have yet to examine how this might be accomplished in our model.

Version Configurations

A requirement that was not addressed at all in this paper is the ability to version groups of classes. For evolutions affecting component classes, it might be convenient to be able to collect a group of classes into a version. This will allow a class to specify the class-versions of its attributes' domains. Such a facility might also assist in the support of the aforementioned complex evolutions.

Programming an adaptation strategy

Our system as described has more versatility than ORION's facility because it supports user-defined instance adaptation information. Consistent with our desire to aid the schema designer, we would like to provide the ability to install user-defined adaptation strategies based on disjoint union data model. For example, a database that must be highly available during business hours could maintain a log of the instances touched during the day and spend the idle overnight hours converting them.

References

[1] Arnold, R. S. **Tutorial on Software Restructuring.** Institute of Electrical and Electronic Engineers. IEEE Society Press, Washington, DC, 1986.

[2] Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases.* in: **Proceedings of the SIGMOD International Conference on Management of Data**, edited by U. Dayal and I. Traiger. San Francisco, CA, 1987.

[3] Bertino, E. 1992. *Personal Communication.*

[4] Bertino, E. *A View Mechanism for Object-Oriented Databases.* in: **Advances in Database Technology – EDBT '92 International Conference on Extending Database Technology**, edited by . Vienna, Austria, 1992. *to appear.*

[5] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. *The GemStone Data Management System.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[6] Casais, E. *Managing Class Evolution in Object-Oriented Systems.* in: **Object Management**, edited by D. C. Tsichritzis. Universite de Geneve, Centre Universitaire d'Informatique, Geneva, 1990, pp. 133–195.

[7] Casais, E. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* Université de Genève, Geneva, 1991.

[8] Delcourt, C. and Zicari, R. *The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System.* in: **Proccedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science**, vol. 512, Springer-Verlag, Geneva, Switzerland, 1991. *A more detailed version is available as [9].*

[9] Delcourt, C. and Zicari, R. *The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System.* Dipartimento di Elettronica Technical Report, no. 91.021, Politecnico di Milano, Milan, Italy, 1991. *A short version of this paper appears in the 1991 ECOOP proceedings.*

[10] Giuse, D. *KR: Constraint-Based Knowledge Representation.* no. CMU-CS-89-142, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, April 1989.

[11] Griswold, W. G. and Notkin, D. *Program Restructuring to Aid Software Maintenance.* Technical Report, no. 90-08-05, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195 USA, September 1990.

[12] Keene, S. E. **Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.** Addison-Wesley, Reading, MA, 1989.

[13] Kim, W., Garza, J., Ballou, N., and Woelk, D. *Architecture of the ORION next-generation database system.* **IEEE Transactions on Knowledge and Data Engineering,** vol. 2 (1990), pp. 109–24.

[14] Kim, W. **Introduction to Object-Oriented Databases. Computer Systems,** MIT Press, Cambridge, MA, 1990.

[15] Lerner, B. S. and Habermann, A. N. *Beyond Schema Evolution to Database Reorganization.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA) and Proccedings of the European Conference on Object-Oriented Programming (ECOOP).** Ottawa, Canada, 1990, pp. 67–76.

[16] Monk, S. and Sommerville, I. *A Model for Versioning Classes in Object-Oriented Databases.* Internal Report, no. SE-91-07, Computing Department, Lancaster University, Lancaster, UK, July 1991.

[17] Motro, A. *Superviews: Virtual Integration of Multiple Databases.* **IEEE Transactions on Software Engineering,** vol. 13 (1987), pp. 785–98.

[18] Skarra, A. H. and Zdonik, S. B. *Type Evolution in an Object-Oriented Database.* **MIT Press Series in Computer Systems,** MIT Press, Cambridge, MA, 1987, pp. 393–415. *An early version of this paper appears in the OOPSLA '86 proceedings.*

[19] Steele, Jr., G. L. **Common Lisp: The Language.** Second Edition. Digital Press, 1990.

[20] Stonebraker, M. *Implementation of Integrity Constraints and Views by Query Modification.* in: **Proceedings of the SIGMOD International Conference on Management of Data.** San Jose, CA, 1975.

[21] Zdonik, S. *Object-Oriented Type Evolution.* in: **Advances in Database Programming Languages.** edited by F. Bancilhon and P. Buneman. ACM Press, New York, NY, 1990, pp. 277–288.

[22] Zicari, R. *A Framework for Schema Updates in an Object-Oriented Database System.* in: **Proceedings of the IEEE Data Engineering Conference.** Japan, 1991. *Short version of [23].*

[23] Zicari, R. *A Framework for Schema Updates in an Object-Oriented Database System.* in: **Building an Object-Oriented Database System: The Story of O_2.** Morgan Kaufmann, 1991. *Also available as Politecnico di Milano, Research Report no. 90-025.*

[24] Zicari, R. *A Framework for O_2 Schema Updates.* Rapport Technique, no. 38-89, GIP Altair, Rocquencourt, France, October 1989.